JaDA – the Java Deadlock Analyzer

Abel Garcia, Cosimo Laneve

JaDA – the Java Deadlock Analyzer

Abel Garcia and Cosimo Laneve

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus
{abel.garcia2,cosimo.laneve}@unibo.it

Abstract. JaDA is a static deadlock analyzer that targets Java bytecode. The core of JaDA is a behavioral type system especially designed to record dependencies between concurrent code. These behavioural types are thereafter analyzed by means of a fixpoint algorithm that reports potential deadlocks in the original Java code. We give a practical presentation of JaDA, highlighting the main connections between the tool and the theory behind it. We also present some of the features for customising the analysis: while the main strength of JaDA is to run in a fully automatic way, user interaction is however possible and may enhance the accuracy of the results. We finally assess JaDA against the current state-of-the-art tools, including a commercial grade one. As one of the main achievements so far, we present the successful analysis of a recursive method that creates a potentially infinite number of threads.

1 Introduction

The present paper describes the prototype implementation of the theory described at [5]

2 Addressing Java bytecode

2.1 Why bytecode?

An important decision we took was not to address the Java language directly because of two reasons: it is quite a complex language and it has no well-defined formal semantics. Therefore we focus instead on the Java bytecode, namely 198 instructions that are the compilation target of every Java application. The bytecode language is quite simple and it has reference semantics that are defined by the Java Virtual Machine behaviour (JVM). Handling the bytecode has also other rather important advantages, for example: the possibility to address other programming languages that are compiled to JVML, like Scala[1], and the possibility to analyze private code in which the source is not available. We have, therefore, defined an inference system that extracts abstract models out of Java bytecodes. This inference system consists of a number of rules which are identical for most of the instructions, these rules mainly differ in the operations that have some effect in the synchronization process, namely invocations, locks acquisition and release, object manipulation and control flow operations.

2.2 Parsing the bytecode

The parsing of the `Java` bytecode is a cumbersome process because of the length of the language syntax. There are several third party tools that aid in this process (e.g. Soot, BCEL, Javassist, etc)[1], some of these libraries provide out-of-the-box handy mechanisms for static analysis: alias analysis, points-to analysis, etc. However, the implementation of `JaDA` we use `ASM`[2] which provides a framework for the bytecode *ast* extraction and manipulation. This framework also provides abstract representations of each one of the bytecode instructions and an abstract implementation of a per-method data-flow analysis. This tool is open source and well documented, which makes easier its adaptation to a particular scenario. We provide more details on the `JaDA` architecture and in Section 6.

2.3 Prerequisites

Although `JaDA` does assumes the existence of a well-formed bytecode generated by the `Java`compiler, `JaDA` requires the bytecode of all dependencies of the code, otherwise there could be parts of the application that remains unanalyzed. Moreover, although the bytecode is not executed, `JaDA` needs to load every existing type to gather some key information in the analysis, like the inheritance information. The loading of the existing types is done dynamically in a sand-boxed class loader to avoid security risks. The full set of dependencies can be specified in `JaDA` through a *classpath*-like configuration.

`JaDA` also assumes that the code targeted by the analysis is free of some of the current limitations of the tool. Currently the tool has two main limitations: (i) the `wait/notify/notify-all` operations are currently not supported, (ii) currently the tool does not consider the concurrent operations constructed using the elements in the `java.util.concurrent` package. There are also other less critical limitations like the analysis of native code and reflection operations, however these ones can be tackled by manually specifying the behavior of the methods involving this limitations.

2.4 Methods resolution

Theoretically our algorithm foresees the analysis of every method in the target code, including those native ones which are considered empty methods if not specified otherwise in the tool settings. However because of efficiency the tool offers the possibility to select a target method, by default the tool chooses the first *main entry point* in the code.

From this point on, the analysis is performed in lazy mode. Starting from the *main*, every method reached by at least one execution path is added to the analysis. Each method is analyzed compositionally, the methods callers are then

[1] See the following link for a full list of existing libraries with this purpose https://java-source.net/open-source/bytecode-libraries

[2] http://asm.ow2.org

queued in to the analysis process whenever the analyzer finds new information about the methods behavior. This is an iterative process, the existence of a fix-point has been proved in [5].

Due to the inheritance, a method could have different implementations depending on the carrier type at runtime. Since our tool works at static time, all possible implementations should be considered in a non-deterministic way. The tool evaluates every possibility, this is a cumbersome process that can affect the performance of the tool in considerable way. However, one of the advantages of the static analysis is that it is relatively not time critical.

In `JVML` there are two main thread operations, the spawning of a thread and the synchronization, this operations are accomplished by the special methods defined by the type `Thread`: `start` and `join`. Such methods are treated in a special way and `JaDA` assumes that these remain unaltered in every children class.

3 Example

A: Do we want to have this section?
A: copy-pasted text from [5]

Figure 1 reports a `Java` class called `Network` and some of its $\texttt{JVML}_d$ representation. The corresponding `main` method creates a network of `n` threads by invoking `buildNetwork` – say $t_1, \cdots, t_n$ – that are all potentially running in parallel with the caller – say t_0. Every two adjacent threads share an object, which are also created by `buildNetwork`.

Assuming that all the threads have a symmetric locking strategy, The `buildNetwork` method will produce a deadlock depending on how it is invoked: when it is called with two different objects it is deadlock-free, otherwise it may deadlock (if also $\texttt{n} > 0$). Therefore, in the case of Figure 1, the program is deadlock free, while it is deadlocked if we comment the instruction `buildNetwork(n,x,y)` and uncomment `buildNetwork(n,x,x)`.

The problematic issue of `Network` is that the number of threads is not known statically – `n` is an argument of `main`. This is displayed in the bytecode of `buildNetwork` in Figure 1 by the instruction at address 30 where a new thread is created and by the instruction at address 37 where the thread is started. The recursive invocation that causes the (static) unboundedness is found at instruction 47.

-.5in-.5in

```
class Network{

 public void main(int n){
   Object x = new Object();
   Object y = new Object();
   // deadlock
   // buildNetwork(n, x, x);
   //no deadlock
   buildNetwork(n, x, y);
 }

 public void buildNetwork(int n,
                Object x, Object y){
   if (n==0) {
     takeLocks(x,y) ;
   } else {
     final Object z = new Object() ;
     Thread thr = new Thread(){
       public void run(){
         takeLocks(x,z) ;
     }} ;
     thr.start();
     this.buildNetwork(n-1,z,y) ;
   }
 }

 public void takeLocks(Object x,
                       Object y){
   synchronized (x) {
     synchronized (y) { }
   }
 }
}
```

```
public void buildNetwork(int n, Object x, Object y)
   0  iload_1          //n
   1  ifne 13
   4  aload_0          //this
   5  aload_2          //x
   6  aload_3          //y
   7  invokevirtual 24 //takeLocks(x, y):void
  10  goto 50
  13  new 3
  16  dup
  17  invokespecial 8  //Object()
  20  astore 4         //z
  22  new 26
  25  dup
  26  aload_0          //this
  27  aload_2          //x
  28  aload 4          //z
  30  invokespecial 28 //Network$1(this, x, z)
  33  astore 5         //thr
  35  aload 5          //thr
  37  invokevirtual 31 //start():void
  40  aload_0          //this
  41  iload_1          //n
  42  iconst_1
  43  isub
  44  aload 4          //z
  46  aload_3          //y
  47  invokevirtual 36 //buildNetwork(n-1, z, y):void
  50  return
```

Fig. 1. Java Network program and corresponding bytecode (only the `buildNetwork` method). Comments in the bytecode give information of the objects used and/or methods invoked in each instruction

The output of `JaDA` for this example is shown on Figure 2.

4 The theory behind

The technique behind `JaDA` is based in the theory of behavioral types, in this case, for every well-typed program we get to known its behavior. Such behavior is an abstraction of the program that grabs the relevant operations regarding the concurrency features of the program, since the typing process is non-deterministic due to the evaluation of all possible execution paths, these behaviors are, in practice, an over-approximation of the program. The soundness of the type system allows to ensure that if the behavioral type of the program is deadlock free so it is the original program.

The typing process is done compositionally in a bottom-up direction, this means that a type has been assigned to every instruction and the type of each method is the composition of the types of the instructions it contains, in the same way, the type of a program is the type of all its methods.

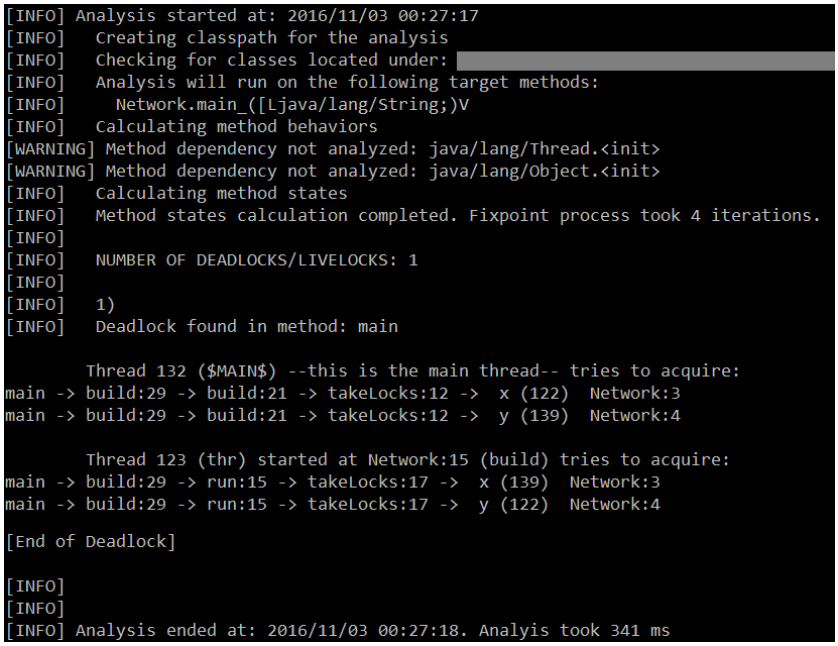

Fig. 2. JaDA analysis output for the Network program

In the following section we highlight some of the details of the JaDA behavioral type system in order to later show its relation with the JaDA implementation. The behavioral type system is fully described in [5].

4.1 Overview of JaDA behavioral type system

Lams. Behavioral types are described in the syntax of *lams* [7] noted ℓ, which express object dependencies and method invocations:

$$\vartheta ::= \top \mid \texttt{int} \mid (a[\overline{\texttt{f}^{\texttt{h}} : \vartheta}], \texttt{C})$$

$$\ell ::= 0 \mid (a, b)_t \mid \texttt{C.m}(a[\overline{\texttt{f} : \vartheta}], \overline{\vartheta}) \to \tau \mid (\nu\, a)\ell \mid \ell \,\&\, \ell \mid \ell + \ell$$

where t ranges over thread identifiers, and a, b range over object identifier. A record type ϑ associates to objects a structural type $a[\overline{\texttt{f} : \vartheta}]$, where a is the identifier of the object and $\overline{\texttt{f} : \vartheta}$ are the fields and corresponding record types.

The type 0 is the empty type; $(a, b)_t$ specifies a dependency between the object a and the object b that has been created by the thread t; $\texttt{C.m}(a[\overline{\texttt{f} : \tau}], \overline{\tau'}) \to \tau''$ defines the invocation of C.m with carrier $a[\overline{\texttt{f} : \tau}]$, with arguments $\overline{\tau'}$ and with returned record type τ''. (The last two elements of the tuple $\overline{\tau'}$ record the thread t that performed the invocation and the last object name b whose lock has been acquired by t. These two informations will be used by our analyzer to build the right dependencies between callers and callees.) The

operation $(\nu\, a)\ell$ creates a new name a whose scope is the type ℓ; the operations $\ell \,\&\, \ell'$ and $\ell + \ell'$ define the conjunction and disjunction of the dependencies in ℓ and ℓ', respectively.

Typing judgments. Let P be a $\texttt{JVML}_d$ bytecode. P will be typed by the judgment[3]:

$$\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, i \vdash_t P : \widehat{\mathcal{T}_i} \,\&\, \widehat{Z_i}^t$$

where:

- BCT: is the a behavioural class table that maps every method of the program to its lam;
- $\Gamma, F, S, Z, \mathcal{T}, K$ are *vectors* indexed by the addresses of P;
- the environment Γ_i maps object names to record types at address i;
- the map F_i takes local variables and returns type values at address i;
- the stack S_i returns a sequence of value types at address i;
- Z_i is the *sequence of object names* locked at at address i;
- $\mathcal{T}_i$ is the *set of thread record types* that are alive at instruction i and that have been created locally either by P or by a method invoked by P
- t is the thread name where P is executed;

The terms $\widehat{\mathcal{T}_i}$ and $\widehat{Z_i}^t$ are a shortened representation of the resulting lam ℓ at instruction i. These terms are expanded in the following way:

$$\widehat{\mathcal{T}_i} = \&_{\vartheta \in \mathcal{T}_i}\, \mathit{typeof}(\vartheta).\texttt{run}(\vartheta, \texttt{C}), t, \mathit{lock}_t)$$
$$\widehat{Z_i}^t = \&_{j \in 2..n}(a_j, a_{j-1})_t$$

Intuitively the first one represents the parallel composition of the `run` methods corresponding to the alive threads in $\mathcal{T}_i$. The second one is expanded into a parallel composition of dependency pairs corresponding to the sequence of locked object names at instruction i. We remark that the sequence Z_i may contain twice the same object name; this means that the thread has acquired twice the corresponding lock. For instance $Z_i = a \cdot a$. In this case $\widehat{Z_i}^t = (a, a)_t$, which is not a circular dependency – this is the reason why we index dependencies with thread names.

Typing ruless. `JaDA` defines typing rules for each one of the bytecode instructions. Figure 4.1 shows (a simplification of) the rules for the `monitor-enter` and `monitorexit` instructions.

The `monitorenter` rule removes the element at the top of the stack and adds it to the set of locked objects. The `monitorexit` rule does the opposite, it removes the element at the top of the stack and removes it from the set of locked objects. We notice that this update has an effect on the lam $\widehat{Z_{i+1}}^t$ by augmenting or reducing the dependencies, respectively.

[3] This judgment is actually a simplification of the actual one

0.3in0in

$$\frac{P[i] = \texttt{monitorenter} \qquad i+1 \in dom(P) \qquad \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \quad Z_{i+1} = a \cdot Z_i \quad \mathcal{T}_i = \mathcal{T}_{i+1}}{\text{BCT}, \Gamma, F, S, Z, \mathcal{T}, i \vdash_t P : \widehat{\mathcal{T}_i} \ \& \ \widehat{Z_i}^t}$$

$$\frac{P[i] = \texttt{monitorexit} \qquad i+1 \in dom(P) \qquad \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \quad Z_{i+1} = Z_i \backslash a \quad \mathcal{T}_i = \mathcal{T}_{i+1}}{\text{BCT}, \Gamma, F, S, Z, \mathcal{T}, i \vdash_t P : \widehat{\mathcal{T}_i} \ \& \ \widehat{Z_i}^t}$$

Fig. 3. JaDA typing rules for locking instructions

4.2 BuildNetwork behavior

Figure **??** show the behavior of the BuildNetwork program as produced by JaDA , the final lams have been simplified for easing the readability. Notice that lam couples $(a, b)_t$ appears as t:(a,b). The commented out part of the method main behavior corresponds to the commented code in the example.

-.3in-.3in [fontsize=,commandchars=
{}] main(this — t,u) = Object.init(x — t,u):x[] + Object.init(y — t,u):y[] +
//buildNetwork(this,, $x, x|t, u$)//$deadlockbuildNetwork(this,, x, y|t, u)$//$no - deadlock$

```
takeLocks(this,x,y | t,u) = t:(u,x) & t:(x,y)

buildNetwork(this,_,x,y | t,u) = takeLocks(this,x,y | t,u) +
    Object.init(z | t,u):z[] +
    Network$1.init(thr, this, x, z | t, z):t1[this$0:this[], val$x:x[], val$z: z[]] +
    Network$1.run(thr | thr,u1) +
    Network$1.run(thr | thr,u1) & buildNetwork(this,_,z,y | t,u)

Object.init(this | t, u):this[] = 0

Network$1.init(this, x1, x2, x3 | t, u):this[this$0:x1, val$x:x2, val$z:x3] = 0

Network$1.run(this[this$0:x1, val$x:x2, val$z:x3] | t, u) = takeLocks(x1, x2, x3 | t, u)
```

Fig. 4. BuildNetwork's lams

The behavior of the main method is straightforward, it contains the invocation to the constructor of the class Object, and the invocation to the buildNetwork method.

In the case of the takeLocks method the behavior is abstracted in the two couples corresponding to the acquisition of the locks of x and y, every couple is formed by the last held lock and the current element. Notice that every method receives an extra argument corresponding to the last acquired lock at the moment of the invocation, in this case that argument is u.

The behavior of the `buildNetwork` method has four states, the initialization to the object `z`, the initialization of the thread `thr`, the launch of the thread `thr` and the recursive invocation. Notice that this last state contains also the running thread `thr`.

The constructor of the class `Object` has an empty behavior.

On the other hand the constructor of the class `Network$1`[4] is more complex. The body of this class accesses to some variables that come from an outer scope (see variables `this`, `x` and `z` in the code), the `JVM` solves this by passing these values to the constructor of the class and assigning them to internal fields. Notice that the behavior of the constructor keeps track of the changes in the carrier object which goes from `this` to `this[this$0:x1, val$x:x2, val$z:x3]` where $\mathtt{x}_i$ are the formal arguments.

Finally, the behavior of the `run` method from the class `Network$1` contains only the invocation to the `takeLocks` method. Notice that `run` method assumes a certain structure from the carrier object.

5 Behavioral types inference and analysis

A: Do we want to have this section?

As stated previously `JaDA` is able to run in a fully automatic fashion, thus not requiring any annotation and being able to *infer* the behavioral types out of the code. The implementation of the type inference mechanism and the further analysis of the resulting behavioral uses two iterative processes.

5.1 Lam and method types inference

The inference of lams and method types from $\mathtt{JVML}_d$ programs is an iterative process that, given a set of method types (which include method effects, such as the structure and the updates of the arguments and the returned object, when applicable, the new threads spawned and the synchronized threads). At each step, it computes the types of method instructions and the method types. The overall process reiterates again if method types are modified, till an invariant is reached. We briefly outline theoretical basis of this solution.

Record types ϑ are equipped with a binary relation $<:$ that is the reflexive and transitive closure of the following pairs:

- $\vartheta <: \top$, and
- $(a[\cdots, \mathtt{f}^{\mathtt{h}} : \vartheta, \cdots], \mathtt{C}) <: (a[\cdots, \mathtt{f}^{\mathtt{h}'} : \vartheta', \cdots], \mathtt{C}')$, if $\mathtt{C} \subseteq \mathtt{C}'$ and, for all fields, $\mathtt{h} \leqslant \mathtt{h}'$ and $\vartheta <: \vartheta'$.

The relation $<:$ defines an upper-semilattice on record types, which allows the computation of least upper-bounds in our algorithm. The relation $<:$ is extended accordingly to functions Γ_i, F_i and S_i Therefore, in the typing rules, like those from Figures 4.1, the equality constraints between Γ_i, F_i, and S_i and

[4] The name *Network$1* is automatically created by the `JVM`for the anonymous type instantiated inside the method `buildNetwork` of the class `Network`

Γ_{i+1}, F_{i+1}, and S_{i+1} are relaxed in favour of $<:$. For instance $\Gamma_i = \Gamma_{i+1}$ becomes $\Gamma_i <: \Gamma_{i+1}$. We also relax identities $\mathcal{T}_i = \mathcal{T}_{i+1}$ into $\mathcal{T}_i \subseteq \mathcal{T}_{i+1}$.

The process typing the instructions uses a data-flow algorithm whose fixpoint is reached when all constraints are satisfied. The pseudo-code of the algorithm is the following:

```
TypeInference(Bct, P, Env_0) -> <Env, L>
1    Env = array[P.size]
2    L = array[P.size]
3    Env[0] = Env_0
4    pending = {0}
5    while( !pending.isEmpty() )
6            index = pending.pop()
7            current = P[index]
8            L[index] = current.type.lam(Bct,Env[index])
9            next_env = current.type.env(Bct,Env[index])
10           if( !current.isReturn() )
11                   if( Merge(next_env, current.next, Env) )
12                                   pending.push(current.next)
13           if( current.isIf() )
14                   if( Merge(next_env, current.jump, Env) )
15                                   pending.push(current.jump)
```

Fig. 5. Pseudo-code of the type inference process for a single method

`TypeInference` starts with the typing environment of the first instruction (see the rules for method definitions) and initialize the set `pending` to $\{0\}$ (only instruction `0` must be typed (see line 4). The typing environment includes Γ, F, S, Z, $\mathcal{T}$ and K of Section 4. The type of every instruction is saved in `L` (line 8). After typing the current instruction we get the typing environment for the next instruction; this environment has to match the current typing environment for the next instruction (line 11). If the two typing environments do not match, they are merged using the $<:$ relation and the corresponding instruction has to be typed again (line 12). Notice that in a first pass all instructions will be typed and their typing environment computed, this is because the initial environment for every instruction (except the first one) is `null`. If the current instruction is an `if` *jump* then the same process is repeated now comparing with the environment at instruction *jump*. The `Merge` function checks if the existing environment is `null`, if so the environment for the current instruction is updated and the method returns true because the instruction needs to be typed again. If an environment for the next instruction already exists then the existing environment is merged with the one just computed, the function `Merge` returns false if the existing typing environment remains unchanged, true otherwise.

The process `TypeInference` terminates because, at every iteration, the environment is augmented or remains unchanged (because of the merge operation).

We remark that our prototype does not uses recursive records, instead those are represented by finite structure. Additionally the upper-semilattice of $<:$ has a finite hight. These premises give necessary conditions for the termination of the above process.

The `TypeInference` process assumes the existence of a BCT. Actually, the computation of the BCT is performed by another iterative process that uses `TypeInference`. The following pseudo code highlights the algorithm:

```
BCT(ClassList) -> <Bct>
1       foreach class in ClassList
2               foreach m in class.Methods
3                       Bct[m] = Empty(m)
4
5       changes = true
6       while(changes)
7               changes = false
8               foreach class in ClassList
9                       foreach m in class.Methods
10                              <Env, L> = TypeInference(Bct, m.P, m.Env_0)
11              changes = changes OR Update(Bct, m, Effects(L))
```

Fig. 6. Inference of the BCT

`BCT` starts by initializing the effects of every method to be empty (line 3). Then every method body is typed with the current `Bct` (line 10) and the typing environments, lams and the method's effects are re-computed from its type (11). This process is repeated until the `Bct` reaches a stable state (line 6). The arguments for the termination of `BCT` are similar to those of `TypeInference`.

5.2 Detection of circularities in lams

Once behavioral types have been computed for the whole $\texttt{JVML}_d$ program, we can analyze the type of the *main* method. The analysis uses the algorithm defined in [7, **?**] that we briefly overview in this section.

First of all, the semantics of lams is very simple: it amounts to unfolding method invocations. The critical points are that (i) every invocation may create new fresh names and (ii) the method definitions may be recursive. These two points imply that a lam model may be infinite state, which makes any analysis nontrivial. Next, we notice that the state of lams are conjunctions ($\&$) of dependencies and method invocation (because types with disjunctions + are modelled by sets of states with conjunctive dependencies).

The results of [7, **?**] allows us to reduce to models of lams that are *finite*, *i.e.* finite disjunctions of finite conjunctions of dependencies. In turn, this finiteness makes possible to decide the presence of a so-called *circular dependency*, namely terms such as $(a, b)_t \,\&\, (b, a)_{t'}$. The reader is referred to [7, **?**] for the details and correctness of the algorithm.

However, the dependencies in `JaDA` are more informative with respect to those from in [7, **?**], here we index each dependency with thread names. There are two reasons for this: (a) in order to cope with `Java` reentrant locks, therefore $(a, b)_t \,\&\, (b, a)_t$ is not a circular dependency (it all happened in the same thread), and (b) in order to avoid to considering circular terms like $(a, b)_t \,\&\, (b, c)_t \,\&\, (a, c)_{t'} \,\&\, (c, b)_{t'}$,

in fact the mutual exclusion on the initial object a makes the concurrent presence of $(b,c)_t$ and $(c,b)_{t'}$ not possible.

The are two differences between the algorithm used in our tool and the one in [7, ?]. The first one is the definition of transitive closure that is the base of the notion of circular dependency. In our case, the transitive closure of $(a,b)_{t_1} \,\&\, (b,c)_{t_2}$ is $(a,b)_{t_1} \,\&\, (b,c)_{t_2} \,\&\, (a,c)_{t_1 \cdot t_2}$, where $t_1 \cdot t_2$ is a sequence of (pairwise) different names that represents a set. For example, the transitive closure of $(a,b)_{t_1} \,\&\, (b,c)_{t_1} \,\&\, (c,a)_{t_2}$ is $(a,b)_{t_1} \,\&\, (b,c)_{t_1} \,\&\, (c,a)_{t_2} \,\&\, (a,c)_{t_1} \,\&\, (a,a)_{t_1 \cdot t_2} \,\&\, (b,a)_{t_1 \cdot t_2}$ This type contains a circular dependency, namely $(a,a)_{t_1 \cdot t_2}$, which is any dependency $(a,a)_{t_1 \cdots t_n}$ with sets $t_1 \cdots t_n$ having at least two elements. The second difference is about the removal of names that are different from the arguments of functions. The algorithm in [7, ?] uses a symbolic computation of lams functions and saturates the models by means of a standard fixpoint technique. In particular, names created by the unfolding process are removed by computing transitive closures of dependencies and projecting them on the arguments of the unfolded function (if there is no circular dependency, otherwise, after the projection, we add a special circularity). We do the same for object names (that are not thread names). On the contrary, fresh thread names are replaced by using two special names – say τ and τ' by means of the following simplification function $\wr \cdot \wr$. Let $\overline{a}$ be the argument names in a method invocation and $\overline{c}$ be the new names created by one unfolding step. Let also $(b,b')_T$, where T is a set of thread names. Then ($\sharp(\cdot)$ gives the cardinality of a set):

$$\wr (b,b')_T \wr \stackrel{def}{=} \begin{cases} (b,b')_{T \cap \overline{a}} & \text{if } \sharp(T \cap \overline{a}) > 1 \\ (b,b')_{\tau} & \text{if } \sharp(T) = 1 \text{ and } T \subseteq \overline{c} \cup \{\tau, \tau'\} \\ (b,b')_{t \cdot \tau} & \text{if } \sharp(T) > 1 \text{ and } \{t\} = T \subseteq \overline{c} \cup \{\tau, \tau'\} \\ (b,b')_{\tau \cdot \tau'} & \text{if } \sharp(T) > 1 \text{ and } T \subseteq \overline{c} \cup \{\tau, \tau'\} \end{cases}$$

As a consequence, abstract models of lam functions are (finite sets of) sets of dependencies using names in the arguments plus τ and τ' (plus a further special name $\varkappa$ – see [7, ?]). Henceforth the finiteness of the models and the decidability of the algorithm.

The pseudo code of the algorithm for computing these abstract models is highlighted here:

`ExpandAndCleanCCT` computes the set of states of every method in the BCT. For each method, the function computes its state (line 5) by instantiating the method invocations in its lam with the models computed in the previous iteration (line 6; at the beginning the model is empty, see line 2). Then the state is simplified as a disjunction of conjunctions (the `Normalize` invocation at line 7). That is the model is a set of elements that are sets of dependency pairs. Every set is then transitively closed (line 9) and then simplified by means of the `Clean` operation in line 10 (which uses the function $\wr \cdot \wr$). After all method states have been recomputed, we check if the new model is different than the old one (line 11). This process goes on until every method has reach an stable set of final states.

```
ExpandAndCleanCCT(Bct)
1    foreach (method in Bct)
2        State[method] = {{}}
3    do
4        changed = false
5        foreach( method in Bct )
6               newState = Replace(State, method.type.lam)
7               newState = Normalize(newState)
8               foreach( set in newState )
9                      set = Transitive_Closure(set)
10                     set = Clean(set)
11              changed = changed OR IsEqual(State[method],newState)
12              State[method] = newState
13   while( changed )
```

Fig. 7. Calculation of the abstract models in the BCT

`JaDA` returns a deadlock if it finds a circularity in the lam of the `main` method. Since the method invocations are stored in the structure implementing the dependency pairs, it is also able to return a trace corresponding to the deadlock.

6 ASM Extension

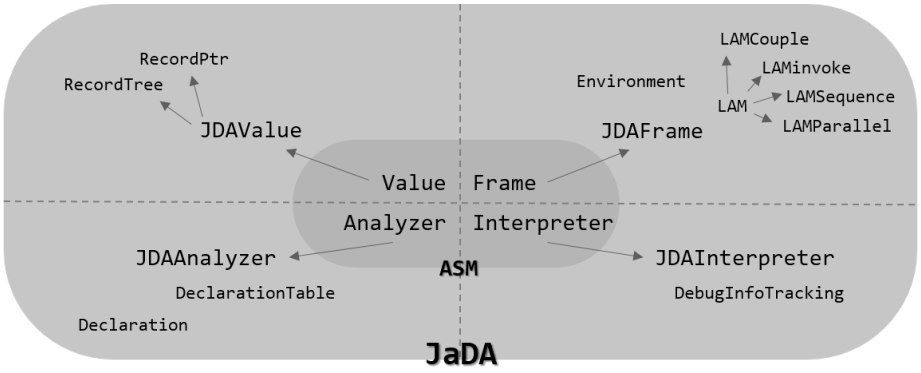

Fig. 8. `JaDA` architecture

The `ASM` framework provides a wide set of tools for manipulating the bytecode. One of the main features of this framework is its support for the implementation of custom bytecode analysis.

There are four main classes oriented to the analysis feature in `ASM`:

- `Value` – this class represents the abstraction of a variable value.
- `Frame` – a frame represents the state the program at the moment of executing a bytecode instruction, namely it contains the state of the stack and the value of the local variables.

– `Interpreter` – the interpreter abstracts the program counter, it operates according to the current instruction updating the values in the current frame.
– `Analyzer` – the analyzer orchestrates the interaction among the elements of the other three classes. For every method the analyzer starts with the empty frame and executes instruction by instruction according to the interpreter. Due to the presence of *jump* instructions, this is not a linear process, for this reason the analysis follows the execution in a *data-flow* fashion, this process stops when each frame in the method has reach a stable state.

The `JaDA`tool has been build on top of the `ASM` framework. The solution is quite rather complex, it contains more than 50 classes and more than 5000 LOC.

The Figure 4 shows part of the class scheme of `JaDA` , in the center of the image the original classes from the `ASM` framework, the arrows denote an inheritance relationship. The added classes are oriented to the instrumentation of the technique so far described.

Values representation. In the `ASM` framework values are conceived as mere names associated with a corresponding type. In `JaDA` just the object identifier and its type is not enough, in our case it is necessary to keep track also of the objects structure (see definition of ϑ). More over in order to abstract the behavior of the `JVM`, `JaDA` presents two type of values: `RecordTree` and `RecordPtr`, these will be used depending on the element storing the values (e.g. the environments Γ_i and methods signature in the BCT will use *RecorTrees* while the local variables F_i and S_i will store *RecordPointers*). The class `RecordTree` is divided in sub-classes matching the different record types categories defined by ϑ. The rest of the main actors in the analysis (the classes for representing frames, the interpreter and the analyzer) have to be changed in order to cope with this variation in the values representation.

Frames. The `JDAFrame` class extends the `ASM Frame` by adding support for all the elements that conform the typing environment, namely structures for Γ_i, F_i, S_i, Z_i and $\mathcal{T}_i$. The `JDAFrame` exposes a method called `execute` which contains the implementation of all of the `JaDA` typing rules, this method relies on the interpreter for symbolically execute the current instruction with the given stack and local variables state. The behavior at every instruction (its *lam*) is finally calculated by means of the implementation of the expressions $\widehat{\mathcal{T}_i}$ & $\widehat{Z_i}^t$ within the context of the corresponding `JDAFrame` object. Another very important method exposed by the `JDAFrame` class is the `merge` method, it is invoked when the analysis process reiterates over an already typed frame. This method implements the logic defined by the operator `<:` , see Section**??**. The decision on whether the subsequent frames should be typed again is taken upon the result of this method.

The interpreter. The `Java` bytecode is a *stack language*, each operation consumes a certain number of elements from the stack and pushes back its result. The `JDAInterpreter` class extends its `ASM` counterpart by adapting to the

JaDA values representation. An important result of JaDA is the possibility to reproduce the traces causing deadlock providing the variable names of the objects involved, the stack trace chain and, when possible[5], the related line numbers in the original code; the process of keeping track of this information is achieved by JDAInterpreter class.

The analyzer. The ASM default analyzer allows to perform a very basic data-flow analysis limited to the scope of a single method. In the same way, JaDA proposes a compositional analysis: the analysis of a method does not go beyond its scope, JDAAnalyzer extends the default ASM implementation by implementing the algorithm described in Figure **??**, this is the building block of the class. In addition JDAAnalyzer generalizes the analysis to the whole program by calculating the final state of the BCT as described by the algorithm in Figure **??**.

Finally the JDAAnalyzer, launches the process to calculate the abstract models of each one of the method declarations in the BCT as described in the algorithm in Figure **??**; and reports the presence of circularities in the main method (or in the method chosen by the user as target of the analysis) .

7 JaDA

7.1 Tool configuration

The main goal of JaDA is to provide a fully automatic tool for the analysis of deadlocks. We note that even though the user interaction (e.g. through code annotations) may allow to gain in precision, this is a cumbersome task and having a tool completely dependent in this sense may lead to a tool only applicable in non-realistic scenarios. However in order to provide some flexibility to the user our tool provides a set of settings that can be used to customize the analysis.

- <target>: this is the only required setting and it is used to specify the target file or folder to analyze. The type of files allowed are: Java class files (".class"), Java jar files (".jar") and compressed zip files (".zip"). In the case of folders, the content of the folder is analyzed recursively.
- verbose[=<*value*>]: the value ranges from 1 to 5, the default and more expressive value is 5.
- class-path <*classpath*>: Standard Java classpath description. If the target contains dependencies other than the ones in the standard library, those should be specified through this property.
- target-method <*methodName*>: fully qualified target method (should be a parameterless void method). Performs the analysis starting from the specified method, if this option is not set, the analysis chooses the first main method found.

[5] This is possible only when the bytecode has been compiled including debugging information

– `additional-targets <classes>`: if `analysis-extent` is set to `custom` this property must contain a comma separated list of the fully qualified names of a subset of classes in the *classpath* to include in the analysis.
– `analysis-extent[=<value>]`:Indicates the extent of the analysis. Possible values are (default value is `classpath`): `full` –analyzes every dependency including system and classpath libraries–, `classpath` –analyzes every library in the classpath–, `custom` –analyzes the classes specified through the property `additional-targets`– and `self` –does not analyze any dependency–.
– `additional-targets <classes>`: if `analysis-extent` is set to `custom` this property must contain a comma separated list of the fully qualified names of a subset of classes in the *classpath* to include in the analysis.
– `custom-types <file>`: a setting file to specify predefined behavioral types.
– `static-constructors[=<value>]`: indicates when the static constructors should be processed, the possibilities are `before-all` and `non-deterministically`. The default option is `before-all`.

7.2 Deliverables

JaDAis available in three forms: a demo website [6], a command line tool (see Figure 2) and an Eclipse plug-in. All three shared the same core, a prototype implementation of the technique discussed in [5]. At the moment of writing this text, the demo website allows only the analysis of single-file programs and a subset of the options previously described. The command line tool and the Eclipse plug-in are available through direct requests. The Eclipse plug-in output includes the display of the execution graph causing the deadlock with links to the source code that originates it (see Figure 5).

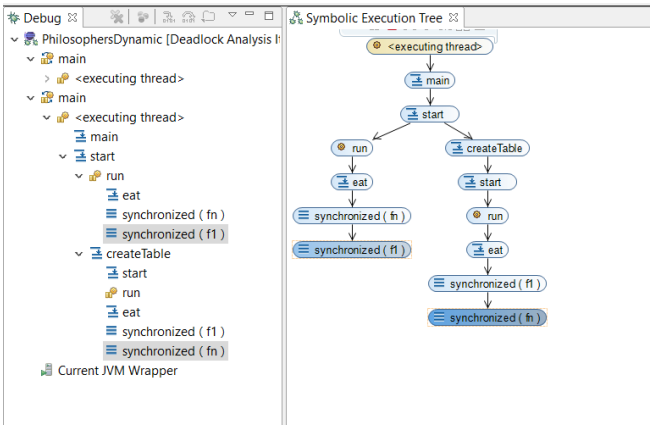

Fig. 9. JaDa Eclipse plug-in screenshot

8 Related tools and assessment

A: copy-pasted text from [5]

JaDA has been compared with respect to the tools that, to the bests of our knowledge, deliver better results for every technique discussed in [5] (see Section "Related Work"). In particular, we have chosen Chord for static analysis [8], Sherlock for dynamic analysis [4], and GoodLock for hybrid analysis [3]. We have also considered a commercial tool, ThreadSafe [6] [2], of which we got the trial version and we don't know much technical details.

Table 1. Comparison with different deadlock detection tools. (Inner cells show the number of deadlocks detected by each tool)

	Static		Hybrid	Dynamic	Commercial
benchmarks	JaDA	Chord	GoodLock	Sherlock	ThreadSafe
Sor	1	1	7	1	4
Hedc	0	24	23	20	11
Vector	0	3	14	4	0
RayTracer	0	1	8	2	0
MolDyn	0	3	6	1	0
MonteCarlo	0	2	23	2	0
Xalan	0	42	210	9	11
BuildNetwork	3	0			0
Philosophers2	1	0			1
PhilosophersN	3	0			0
StaticFields	1	1			1
ScalaSimpleDeadlock	1				
ScalaPhilosophersN	3				

We report our analysis in Table 1. The source of all benchmarks is available either at [4, 8] or in the JaDA-deadlocks repository[7]. Out of the four chosen tools, we were able to install and effectively test only two of them: Chord and ThreadSafe; the results corresponding to GoodLock and Sherlock come from [4]. We also had problems in testing Chord with some of the examples in the benchmarks, perhaps due to some miss-configurations, that we were not able to solve because Chord has been discontinued.

It is worth to notice that JaDA detects only one deadlock in the benchmarks of the first block, thus returning less false positives than the other ones. The second block corresponds to examples designed to test our tool against complex deadlock scenarios like the Network program. We notice that even commercial grade tools like ThreadSafe fail to detect those kinds of deadlocks. While the results in Table 1 are encouraging, we will deliver more convincing ones as soon as JaDA will be integrated with the wait-notify-notifyAll synchronization mechanisms – see below. In addition we report two examples of Scala programs,

[6] http://www.contemplateltd.com/threadsafe
[7] https://github.com/abelunibo/Java-Deadlocks

these programs have been compiled with the version 2.11 of the `Scala` compiler. We remark that, to the best of our knowledge at the moment of writing this, there were no static deadlock analysis tools for such language.

9 Limitations and ongoing extensions

A current limitation of `JaDA`is the lack of support for the synchronizations operations `wait`, `notify` and `notifyAll`. These methods enable programmers to access the `JVM`'s support for expressing thread coordination. They are public and final methods of the class `Object`, so they are inherited by all classes and cannot be modified. The invocations to `wait`, `notify` and `notifyAll` succeed for threads that already hold the lock of the object a on the stack. In this case, the `wait` instruction moves its own thread to the *wait set* of the object a and the object is relinquished by performing as many unlock operations as the integer stored in the lock field of a. The instructions `notify` and `notifyAll` respectively wake up one thread and all the threads in the wait set of o. These threads are re-enabled for thread scheduling, which means competing for acquiring the lock of a. The winner will lock the object a as many unmatched `monitorenter` it did on a before the `wait`-operation.

The *wait-notify* relation between threads can easily lead to deadlocks. There are two possible scenarios:

1. the `o.wait()` operation in t does not *happens-before* a matching `o.notify()` in t';
2. a lock on an object (different than `o`) held by t is blocking the execution of t', thus preventing the invocation of `o.notify()` to happen.

The solution we are investigating uses *lams* with special couples $(a, a^w)_t$ and $(a, a^n)_t$ for representing the wait-notify dependencies. For example, for the wait operation, $(a, a^w)_t$ can be read as "thread t has the lock of `a` and has invoked method `wait` on it". Similarly $(a, a^n)_t$ means "thread t has the lock of `a` and has invoked method `notify` on it". However, these dependencies are not sufficient for deadlock analysis. In fact, the mere presence of a wait couple indicates a potential deadlock. For example a lam like

$$\ell = (a, a^w)_t \ \& \ (a, a^n)_{t'}$$

has to be considered as a deadlock, since nothing ensures that the notification will happen after the wait invocation (in this case it will depend on the scheduler). In any case, this would be a huge over-approximation. The precision can be enhanced if it is guaranteed that the wait operation *happens-before* the matching notification. For this purpose we keep track of one more type of lam couples: $(a, t')_t$ to be read as "thread t' was spawned by thread t while it was holding the lock on a". Next, consider the following states:

$$\ell_1 = (a, a^w)_t \ \& \ (a, t')_t \ \& \ (a, a^n)_{t'} \qquad \ell_2 = (b, a)_t \ \& \ (a, a^w)_t \ \& \ (b, a)_{t'} \ \& \ (a, a^n)_{t'}$$

We can draw the following conclusions. First, ℓ_1 is not deadlocked, when t' started, t is holding the lock of a, so the notification will not occur until t releases the lock of a, which happens exactly when the `wait` is invoked. Second, ℓ_2 is deadlocked, the situation is similar to the previous one, but after releasing a the thread t still holds the lock on b. The thread t', on the other hand, cannot grab the lock on a until it can grab the lock on b. This implies that the notification is never performed.

Following this analysis, we can define a strategy for deciding whether a wait couple is matched by a notify couple:

> *a lam that contains* $(a, a^w)_t$ & $(a, a^n)_{t'}$ *is safe whenever (i)* t' *is spawned by* t *while holding the lock on* `a` *and (ii) whenever* a *is the only object in common in the lock chains acquired by* t *and* t' *before the wait and the notify operations, respectively.*

We are currently studying the soundness of this solution and we will report our results in a future work.

10 Conclusions

`JaDA` is a static deadlock analysis tool that targets the the `JVM` bytecode, allowing the analysis of any well-compiled `Java` program, as well as, programs written in other languages that are also executed within the `JVM` like `Scala`. The technique underlying `JaDA` is based on a custom behavioral type system that abstracts the main features of the programs with respect to the concurrent operations.

The tool is designed to run in a totally automatic fashion, meaning that the inference of the program behavioral type and the subsequent analysis could be done unassisted. User intervention is, however, possible and may enhance the precision of the program, for example, in presence of native methods.

Even though the tool is still under development, we have been able to asses it by analyzing a set of `Java` and `Scala` programs. We have presented a comparison between the results of `JaDA` and some of the existing deadlock analysis tools, among which is a commercial grade one. The results obtained so far are very promising and we expect to gain more precision as the development continues.

The next steps for `JaDA`include adding support for the synchronization operations expressible by means of the `wait`, `notify` and `notifyAll` methods. This chapter concludes with a notion on how to tackle these.

References

1. The scala language. 2016.
2. Robert Atkey and Donald Sannella. Threadsafe: Static analysis for java concurrency. *ECEASST*, 72, 2015.
3. Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *in Hardware and Software Verification and Testing*, volume 3875 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2005.

4. Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE-22)*, pages 353–365. ACM, 2014.
5. Abel Garcia and Cosimo Laneve. Deadlock detection of java bytecode. submitted, 2016.
6. Abel Garcia and Cosimo Laneve. `JaDA` – the `Java` Deadlock Analyzer. Available at `JaDA.cs.unibo.it` from October 28, 2016.
7. Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of 25th International Conference on Concurrency Theory CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.
8. Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE 2009)*, pages 386–396. ACM, 2009.